MASTER'S THESIS | LUND UNIVERSITY 2018

Machine-learning-assisted scene detection

Tony Ngo, Axel Bojrup

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-40

Machine-learning-assisted scene detection

Tony Ngo
mas13tng@student.lu.se

Axel Bojrup
dat15abo@student.lu.se

September 6, 2018

Master's thesis work carried out at Sony Mobile.

Supervisors: Pierre Nugues, pierre.nugues@cs.lth.se
Sebastian Raase, sebastian.raase@sonymobile.com

Examiner: Jacek Malec, jacek.malec@cs.lth.se

Abstract

In this thesis, we explored the possibility of using machine learning to do scene recognition on a mobile device. In order to train our scene recognition model, through machine learning, we first created a dataset. Then, we deployed the model on a mobile device. The deployed model takes as an input a preview from the camera sensor, and outputs a scene. Our model was created using convolutional neural networks (CNN). We investigated different CNN architectures which we will present evaluations of. We demonstrate that it is feasible to use CNN to do scene recognition on a mobile device. With a CNN architecture called MobilenetV1-1, we achieved an F1-score of 86.9%. We used an ensemble method, where the collective result achieved a higher F1-score than what any of the networks did separately. Runtime results for inference of various architectures on Sony Xperia Z3 are also presented, ranging from 41 ms to 2750 ms.

Keywords: machine learning, convolutional neural network, mobilenets, scene recognition

Acknowledgements

We would like to thank Pierre Nugues for his guidance and helpful insights throughout this thesis. We would also like to thank Sebastian Raase at Sony Mobile for his valuable feedback on our thesis and discussions of project progression. Lastly, thanks to the Sony Mobile camera team for help and input throughout the project and Nercivan Mahmudovska and Sony Mobile for making this project possible.

Contents

1 Introduction **7**
1.1 Problem definition . . . 8
1.2 Contributions . . . 9
1.3 Related work . . . 9

2 Background **11**
2.1 Scene Recognition . . . 11
2.2 Data representation of an image . . . 11
2.3 Neural network . . . 11
2.3.1 Learning algorithm . . . 12
2.3.2 Hyperparameters . . . 13
2.3.3 Activation function . . . 14
2.3.4 Convolutional Neural Network . . . 16
2.4 Architectures . . . 21
2.5 Transfer learning . . . 24
2.6 Ensemble learning . . . 24
2.7 Framework . . . 25
2.8 Data sources . . . 27

3 Metrics **29**
3.1 Confusion Matrix . . . 29
3.2 F1-Score . . . 29
3.3 Evaluation graphs . . . 30
3.4 Underfit and Overfit . . . 31
3.5 Stratified sampling . . . 31

4 Method **33**
4.1 Model . . . 33
4.2 Preprocessing . . . 33
4.3 Experiment . . . 34

4.4 Android app 34
4.5 Data 35

5 Results **39**
5.1 Comparison of networks 39
5.1.1 InceptionV4 39
5.1.2 MobilenetV1 42
5.2 Dataset modifications 44
5.3 ADAM and fine tuning 44
5.4 Places dataset 44
5.5 Learning curve 46
5.6 Ensemble learning 47

6 Discussion **49**
6.1 Comparison of networks 49
6.2 Dataset modifications 49
6.3 ADAM and fine tuning 50
6.4 Places dataset 50
6.5 Learning curve 50
6.6 Ensemble learning 50

7 Conclusions & Future work **53**
7.1 Conclusions 53
7.2 Future work 54

Bibliography **55**

Chapter 1
Introduction

Scene recognition is a computer vision task which, given an image, will label that image as a scene. Consider Figure 1.1, where the left image represents a beach scene and the right image represents a snow scene. The task is then to construct a scene recognition model which labels the images accordingly. This comes natural to humans and is a rather effortless task, but for computers it is a more challenging task (Ballard and Sabbah, 1983). Computer vision has been through a breakthrough in recent years and is now one of the most active research areas for deep learning applications (Goodfellow et al., 2016).

Automatic image captioning is another computer vision task (See Figure 1.2). For this task, a system is provided with an image which then will be annotated with an appropriate caption. This system learned how various objects in an image related to each other by a detailed accompanied caption for each image. This is closely related to a more traditional

(a) A beach scene.

(b) A snow scene.

Figure 1.1: Two images with their corresponding scene label.

(a) "man in blue wetsuit is surfing on wave."

(b) "baseball player is throwing ball in game."

Figure 1.2: Two captions generated by Karpathy and Fei-Fei's model and their corresponding image.

computer vision task, namely object recognition, where an given image is labelled for each object in the image (Karpathy and Fei-Fei, 2017).

The idea of this thesis is to create a scene recognition model through deep learning which will be deployed to a mobile phone. The deployed model should only be concerned with inference, thus all learning will be performed on an separate machine. The model should be able to access the preview image from the camera, which will be used as an input to the scene recognition model. We will not investigate any application for the created model in this thesis. One application could be to map the scene returned from the model to a pre-defined camera parameter setting.

1.1 Problem definition

The goal of this work is to explore the possibility of using machine learning, and in particular, neural networks to do scene detection. The focus will be to determine what currently available architectures will be most suitable to run on a mobile device. That is, in this thesis we will not construct any new network architectures, we will only evaluate already existing publicly available architectures. By investigating different architectures, we want to make evaluations based on number of parameters, speed and accuracy. Our research questions are the following:

- What existing neural network architecture works best on a mobile device?
- What is the relation between the number of parameters, speed and accuracy?

1.2 Contributions

In this project we have created a scene recognition model. We have evaluated some state-of-the-art architectures on a new dataset which we created. We also tried an ensemble method - stacking which, as expected, improved the result.

In this project most of the work was equally divided. The only thing that required individual work was when we were working with the data. This required us to create our own scripts for data management.

1.3 Related work

Interest and development in computer vision, especially in deep learning, have been increasing as it looks promising for the problems that have been impractical to handle before. During this decade, research in deep learning has made huge progress. Advancements in hardware performance, GPUs in particular, played a big part in making deep learning faster and more accessible.

A major breakthrough came in 2012 when Krizhevsky et al. presented AlexNet, a large, deep convolutional neural network (CNN) that won the ILSVRC-2012 competitors (ImageNet Large Scale Visual Recognition Challenge). It achieved a top-5 test error rate of 15.3%, compared to the 26% achieved by their competition. The paper found that using the ReLU instead of hyperbolic tangent for nonlinearity function decreased training time. It also used data augmentation and dropout and showed that it prevented overfitting.

In 2015a, Szegedy et al. introduced GoogLeNet (Inception architecture), a 22-layer CNN which won the ILSVRC-2014 with top 5 error rate of 6.7%, performing close to humans of 5.1% (Andrej Karpathy). This network used 12x fewer parameters than AlexNet and used what is called Inception modules, which are explained further in Section 2.4.

Recent research has been done in deep learning for mobile vision applications. There exists a family of architectures called MobileNet, which have shown promising results for Mobile applications (Howard et al., 2017). That work was revised in Sandler et al. (2018), resulting in even smaller networks and with higher accuracies.

Many of these networks have performed well on the ImageNet dataset which is an object-centric dataset. Zhou et al. showed in 2017 that networks based on object-centric data will have different internal representations than networks based on scene-centric data (Zhou et al., 2014, 2017). Object-centric data is defined as images that contains an object in this case. Accordingly, scene-centric data is defined as images that contain scenes, consistent with the description of scenes that was provided earlier.

Chapter 2
Background

2.1 Scene Recognition

Scene recognition is a part of computer vision. It might be as useful to know what scene a particular object is present in as what object it is. For instance, a table (object) in a kitchen (scene) serves another purpose than what a table does in a classroom. Applying deep learning solutions to such problems has proven promising. In this chapter will we present key constituents to deep learning.

2.2 Data representation of an image

In computing, images are generally represented as arrays of numbers. One such representation uses a 3-dimensional array where the dimensions refer to the three color channels red, green blue (RGB). Each number in the array will have a value ranging from 0 (byte) to 255 (byte). For instance, a 100x100 pixel image will contain 30'000 ($100 \cdot 100 \cdot 3$) numbers.

2.3 Neural network

A neural network is a model used in machine learning which has references to neuroscience. It is generally arranged in *layers*, where each layer consists of a number of *neurons*. There is an input layer and an output layer. In a feedforward neural network, the neurons are connected in a directed acyclic graph. A single-layer feedforward network consists of an input layer and an output layer, while a multi-layer feedforward network consists of least one layer called hidden layer between the input and output layer. A neuron receives input from other neurons and computes its own value. Neurons in a layer are connected to the neurons of the next layer (feedforward network). Each connection to a

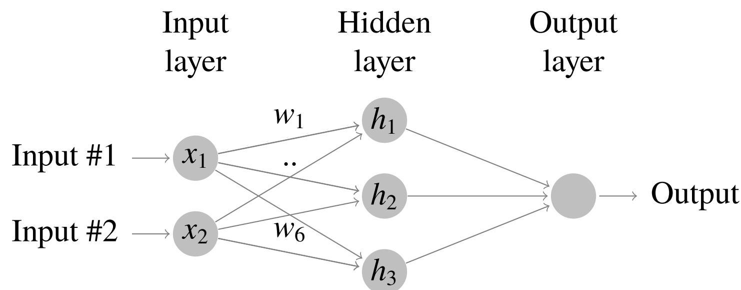


Figure 2.1: A neural network with input layer, output layer and one hidden layer

neuron has a weight (also called *parameter*) which is adjusted as the network is training. Training means that finding the weights that the network will correctly map the input to the desired output. The output h of a hidden layer is calculated as:

$$h = W^T x + b, \tag{2.1}$$

where W is the weight matrix, x is the input and b is a bias term. An example of a feed-forward network is shown in Figure 2.1. The figure also shows that neurons in the hidden layer have connections to all the neurons in the previous layer. Such hidden layers are called *fully connected layers*. In the example, the input x would be a vector of size 2x1, the weight vector W of size 3x2 and the bias b would be of size 3x1. Applying Equation 2.1 would then result in a vector h of size 3x1.

2.3.1 Learning algorithm

The training of a convolutional neural network is based on 3 concepts, which are: *loss function*, *back-progation algorithm* & *optimization*. Through the various transformations employed by the different network layers, information is propagated forward in the network (forward propagation). While training a network, this phase is carried out until the *loss function* can be computed. The information from the loss function will propagate backward through the network, to calculate the gradient. Using this gradient an *optimization* is performed, typically by a *stochastic gradient descent algorithm*, or a variant of it.

Loss function

A loss function is used to measure the quality of the network output. It compares the predictions with the true labels. The lower the loss, the better the model will classify the training data. Depending on application, this is could be done by a function called cross-entropy, which can be intepreted as a notion of difference between two distributions and is shown in 2.2 (Goodfellow et al., 2016).

$$H(P, Q) = -\mathbb{E}_{x \sim P} log Q(x) \tag{2.2}$$

Back-propagation

The chain-rule of calculus is used to calculate the derivative of function compositions (analogous to layers in succession in a neural network). Back-propagation is an algorithm that computes the chain-rule of calculus efficiently. In more general terms, it is used to calculate the gradient, which is used for optimization (Goodfellow et al., 2016).

Optimization

In the context of neural networks, an optimizer is trying to find the weights of the network that minimizes the loss function. Two different optimizers have been used during the project. These two optizimers are based on the stochastic gradient descent and have shown to perform really well.

- **RMSProp**: RMSProp is an optimization method where each of the weights has a learning rate. The learning rate is divided by a running average of the squares of the previous gradients for that particular weight (Tieleman, 2018):

  $$MeanSquare(w, t) = 0.9MeanSquare(w, t - 1) + (0.1)(\nabla Q_i(w))^2$$

  The parameter/weight is updated:

  $$w = w - \frac{\eta}{\sqrt{MeanSquare(w, t) + \delta}}(\nabla Q_i(w))$$

  RMSProp can converge faster by using the exponentially decaying average to remove history from the extreme past. An advantage of RMSProp is that it works well in non-stationary settings (Kingma and Ba, 2014).

- **ADAM**: Adam is another optimization method in which the adaptive learning rate is calculated for each parameter. It is related to both RMSProp and momentum. Momentum is used to accelerate learning, particularly when there exist noisy gradients, high curvature or small but consistent gradients. Momentum accumulates an exponentially decaying moving average of past gradients and move in their direction which dampen oscillations. Adam computes the exponential moving average of the gradient (estimate of first order moment) and squared gradient (similar to RMSProp). However, ADAM uses bias-correction terms to the first-order moments and the second-order moments which are missing in RMSProp. It could cause the second-order moment estimate of RMSProp to have high bias early in training due to not having the correction term (Goodfellow et al., 2016). An advantage of ADAM is that works well with sparse gradients, due to the bias correction.

2.3.2 Hyperparameters

The hyperparameters that are described in this section are defined as variables that are set prior to training. These parameters are not part of the model, but instead affect the training process.

Batch size

Batch size is the subset of the dataset that is used when doing one update to the model parameters. A recommendation of a batch size is between 2 and 32 training samples (Bengio, 2012).

Epoch

Epoch is the number of times the whole training set has been passed in to the network when training.

2.3.3 Activation function

The activation function is a transformation that is performed on the output of a layer, to induce the network to nonlinearities, which is important for the network. If this property would be omitted, then every layer would be a linear transformation, and the network as a whole would just be a linear transformation. Hence it won't be able to capture all the complexities of the given problem the networks tries to solve. The output of an activation function on a hidden layer is referred to as a hidden unit. There are several commonly used activation functions, which are shown in Equation 2.3 and 2.4:

$$g_1(z) = \sigma(z) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

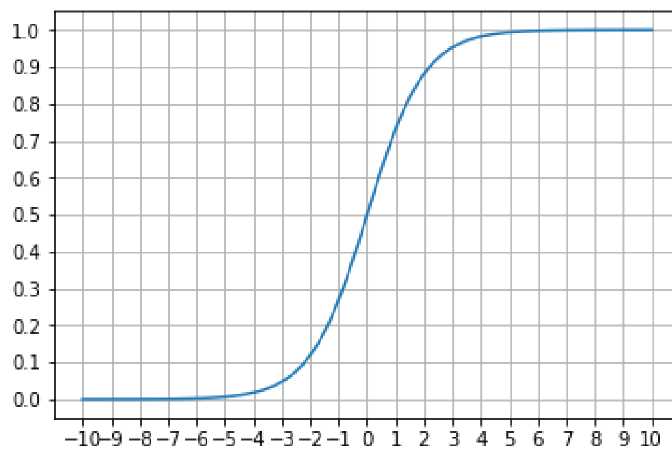


Figure 2.2: Plot of Logistic function, from Equation 2.3

$$g_2(z) = tanh(z) = 2\sigma(2z) - 1 \tag{2.4}$$

$g_1(z)$ is called the *logistic function* and $g_2(z)$ is the closely related *hyperbolic tangent* (See visualization in Figure 2.2 and 2.3 respectively). Both functions saturate across parts of their domain, which makes gradient based learning difficult. For this reason attention has shifted to Rectified Linear Units (ReLU), $g_3(z)$, shown in Equation 2.5 and visualized in Figure 2.4.

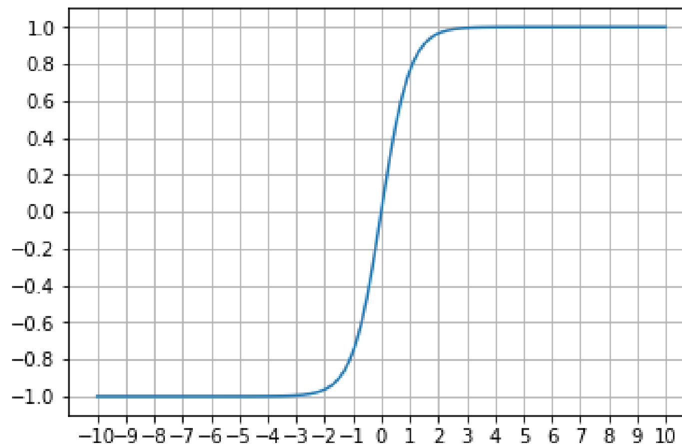


Figure 2.3: Plot of Hyperbolic tangent, from Equation 2.4

$$g_3(z) = max\{0, z\} = \begin{cases} 0, z < 0 \\ z, z \geq 0 \end{cases} \tag{2.5}$$

The ReLU activation function won't suffer from saturation. It is also very similiar to a linear function, which makes it easier to optimize.

The *softmax function* $g_4(z_i)$, shown in Equation 2.6, is commonly used on the output of the last layer for a classifier to provide the probability distribution of the various classes (Goodfellow et al., 2016).

$$g_4(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2.6}$$

This function returns values in the range $0 < g_4(z_i) \leq 1$ with $\sum_i z_i = 1$. See following example, where $z = [1, 2, 3, 4, 1, 2, 3]$:

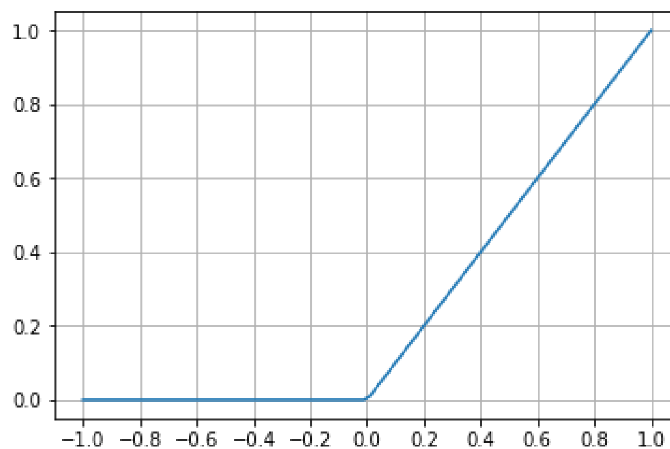


Figure 2.4: Plot of Rectified Linear Unit, from Equation 2.5

$$g(z_0) = g(1) = \frac{e}{e + e^2 + e^3 + e^4 + e + e^2 + e^3} \approx 0.024$$

$$\vdots$$

$$g(z_6) = g(3) = \frac{e^3}{e + e^2 + e^3 + e^4 + e + e^2 + e^3} \approx 0.175$$

$$g(\mathbf{z}) = [0.024, 0.064, 0.175, 0.475, 0.024, 0.064, 0.175]$$

2.3.4 Convolutional Neural Network

Deep Learning makes it possible for models that consist of multiple layers to learn representations of data with multiple levels of abstraction (LeCun et al., 2015). "Deep" refers to the depth of the network; the number of hidden layers representing the model. The layers are describing different functions that form the network (Goodfellow et al., 2016). A deep neural network is one of many different types of networks used in deep learning. A Convolutional Neural Network (CNN) is one type of deep neural network. In addition to using multiple hidden layers, there are four concepts of using a CNN, namely *convolution layers*, *parameter sharing*, *sparse connections* and *pooling*. A convolutional neural network uses an operation called convolution instead of a general matrix multiplication (Equation 2.1) in at least one of the layers.

Convolutional Layer

The core block of a convolutional neural network is the convolutional layer which uses a convolution operation. Convolution is a mathematical operation on two functions to produce a third function. In CNNS, the convolution operation is applied on the input data of the layer with a *filter* (or kernel). This is done by sliding the filter along the width and height of the entire input. The output is called feature/activation map. The output size is controlled by three hyperparameters: depth, zero-padding and stride.

- The depth corresponds to the number of filters.
- Zero padding is whether the input is padded with zeros around the border.
- Stride refers to how many pixels a filter is moved per step.

The process can be seen in Figure 2.5. The convolution layer computes the convolution between the input data X and the filter W (consisting of weights) and creates an activation map (X∗W). Each step the filter is moved leads to an activation of the neuron on that specific location of the input. The output is stored in an activation map. The figure shows that zero padding and a stride of 1 were used. Note that if stride was 2, the output matrix X∗W would have been of size 3x3.

Depending on the input and the hyperparameters to a convolution layer, the convolution could be performed with different filters, where the output of the layer would then be a stack of activation's maps.

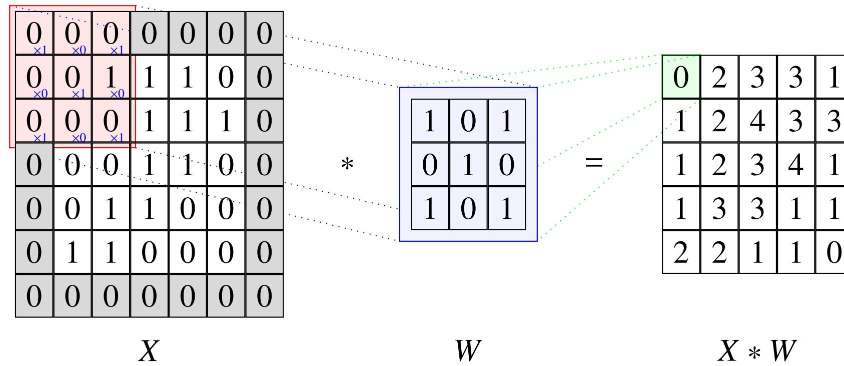
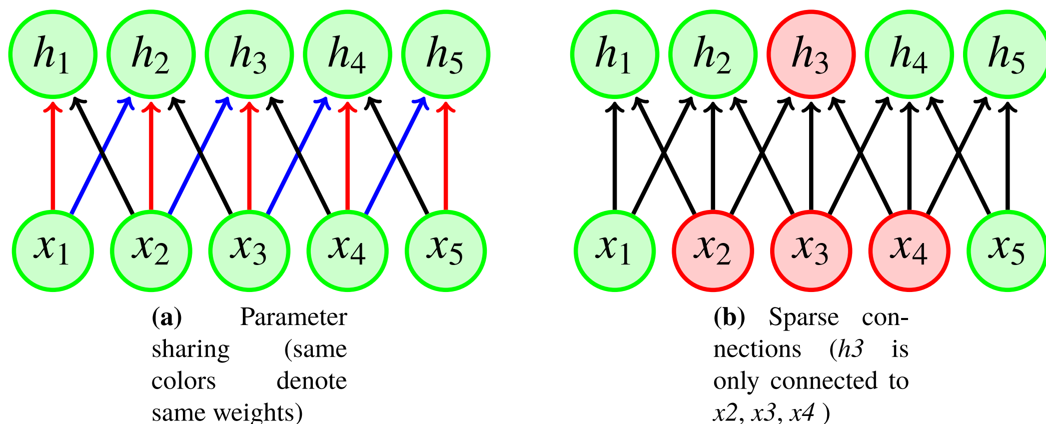


Figure 2.5: Example how a convolution is performed. The grey cells are added due to zero padding.



(a) Parameter sharing (same colors denote same weights)

(b) Sparse connections (*h3* is only connected to *x2*, *x3*, *x4*)

Figure 2.6: Concepts in a CNN

The input size of a convolutional layer is thereby a D_F x D_F x M, where D_F is the width and height (assuming the input is a square) of the input and M is the number of input channels. The convolutional filter K is of size D_K x D_K x M where D_K is the size (assuming it is a square) of the filter and M is the number of input channels (see Figure 2.7 a). The output size of the convolutional layer is D_G x D_G x N where N is the number of output channels (number of filters/activations maps) and D_G is the width and height of the output. The number of add and multiply operations (referred to operations henceforth) of standard convolutions in terms of number of multiplications are therefore:

$$D_G \cdot D_G \cdot D_K \cdot D_K \cdot M \cdot N = D_G^2 \cdot D_K^2 \cdot M \cdot N \tag{2.7}$$

Parameter sharing

A convolution network uses parameter sharing which refers to using the same parameter for more than one function (Goodfellow et al., 2016). In a classic neural network, weights are independent. Parameter sharing reduces the number of independent parameters, thus allowing for faster implementation. Figure 2.7a shows an example of parameter sharing, where edges of same color share weights.

Sparse Connectivity

In a regular neural network, every output neuron is interacting with every input neuron (fully-connected layer). In a convolution layer, nodes often only interact with a few other neurons because of convolution. These neurons are called the receptive field of the output node. Having sparse connections means that fewer operations are required to calculate the output (Goodfellow et al., 2016). Figure 2.7b shows that only the inputs x_2, x_3 and x_4 are connected to the output neuron h_3.

Depthwise Separable convolution

Depthwise separable convolution is another type of convolution which is used in deep learning. It combines a depthwise convolution and a pointwise convolution. The depthwise separable convolution is divided into two layers, one layer responsible for applying a single filter to each input channel (depthwise convolution) and the other combines the outputs of the depthwise convolution by applying a 1x1 convolution (pointwise convolution). This technique is applied instead of a regular convolution due to the reduction in the number of operations and model size (Howard et al., 2017). The following can be divided in two different stages:

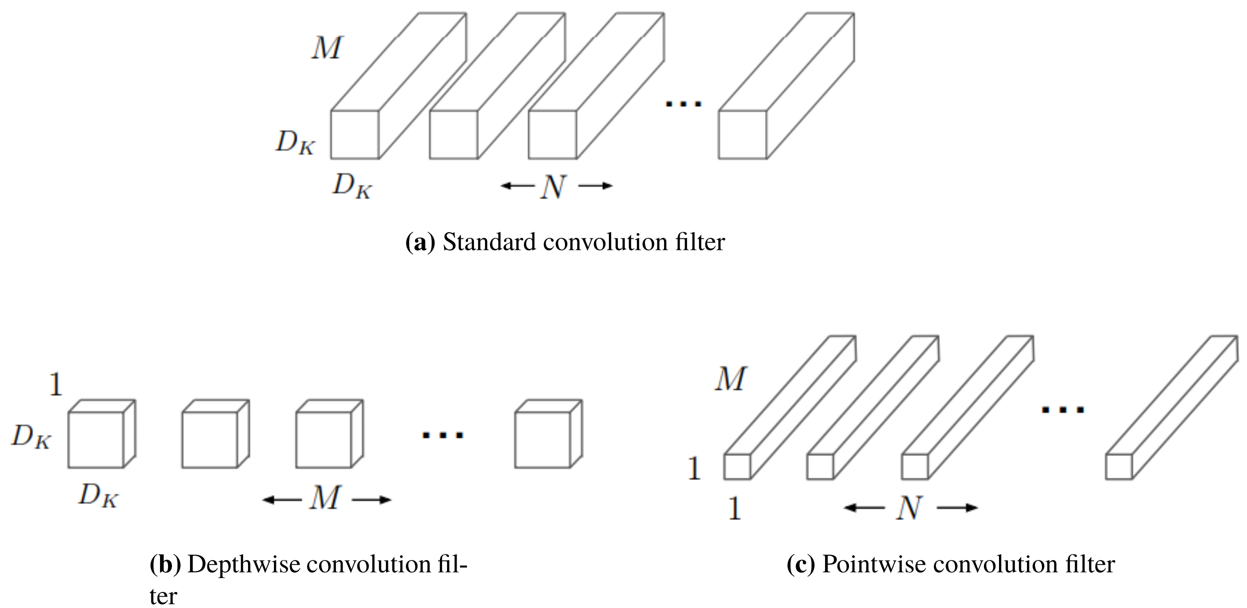
- For depthwise convolution: The difference between a standard and a depthwise convolution is that in depthwise convolution, it applies a single filter to one input channel whereas in the standard convolution, a filter is applied over all input channels. A depthwise convolution filter of size D_K x D_K will yield the number of operations $D_K \cdot D_K \cdot D_G \cdot D_G$ when applied to the mth input channel where D_G is the width and the height of the output (see Figure 2.7 b). When applied to M number of filters, the output size of depthwise convolution is D_G x D_G x M and the total number of operations is:
$$D_K \cdot D_K \cdot D_G \cdot D_G \cdot M \tag{2.8}$$
- For pointwise convolution: The previous step filtered the input channels but did not combine them to create new features. This step called pointwise convolution computes a linear combination of the output of the depthwise convolution by using a 1 x 1 convolution. A convolution filter of size 1 x 1 x M (Shown in Figure 2.7 c)), where M is the number of input channels (the number of channels from the output of the depthwise convolution) is applied to the output of the previous step which will give the new and final output of size D_G x D_G x N where N is number of 1 x 1 filters. The total number of operations for pointwise convolution is:
$$D_G \cdot D_G \cdot M \cdot N \tag{2.9}$$

Depthwise separable convolution number of operations is therefore:

$$D_K \cdot D_K \cdot D_G \cdot D_G \cdot M + D_G \cdot D_G \cdot M \cdot N = M \cdot D_G^2(D_K^2 + N) \tag{2.10}$$

which is the sum of depthwise convolution and pointwise convolution. The relation between depthwise separable convolution and standard convolution can be written as:

$$\frac{M \cdot D_G^2(D_K^2 + N)}{M \cdot D_G^2(D_K^2 \cdot N)} = \frac{1}{N} + \frac{1}{D_K^2} \tag{2.11}$$



(a) Standard convolution filter

(b) Depthwise convolution filter

(c) Pointwise convolution filter

Figure 2.7: The different types of convolution filters. Images taken from MobilenetV1 paper

For example, if D_K=3 and N=512, it would give the ratio 1/512 + 1/9= 0.111 which means depthwise separable convolution has around 9 times fewer operations than the standard convolution.

Pooling layer

A pooling layer is responsible for reducing (downsampling) the spatial dimensions (width and height) of each activation/feature map. Pooling can be performed by different operations such as max pooling or average pooling, where max pooling is the most common. The pooling layer needs two hyperparameters: the size of the filter F and the stride S. A common shape of a pooling layer is with filters of size 2x2 with stride 2 using max pooling. For this setup, the filter will then slide by 2 cells along the width and height and calculating the max value (max pooling) over the 2x2 region (4 numbers). Average pooling would instead calculate the average value. It will produce an output of size W x H x D where W and H are the new width and height in which 75% of the activations have been removed. D is the same depth as the input (see Figure 2.8). Pooling has the advantage of reducing the number of parameters and reduce the computation in the network and also decreases the risk of overfitting (see Chapter 3) (Karpathy, 2018).

Dropout

Applying a dropout to a unit in a network means you give that unit a probability to be dropped. Consider Figure 2.9, where this is visualized, the $x1$ unit will be kept with a probability μ, otherwise it will be multiplied with 0. Thus being droppped, which is one way of utilizing dropout. If dropout is used, it is used during training. When the model is

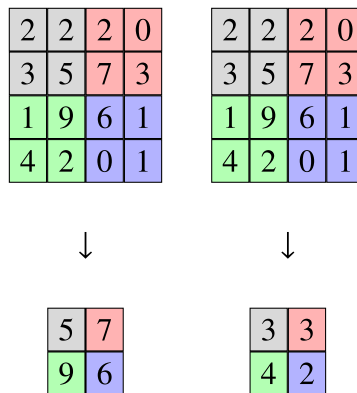


Figure 2.8: An example of max and average pooling. Filter size is 2x2 with stride 2

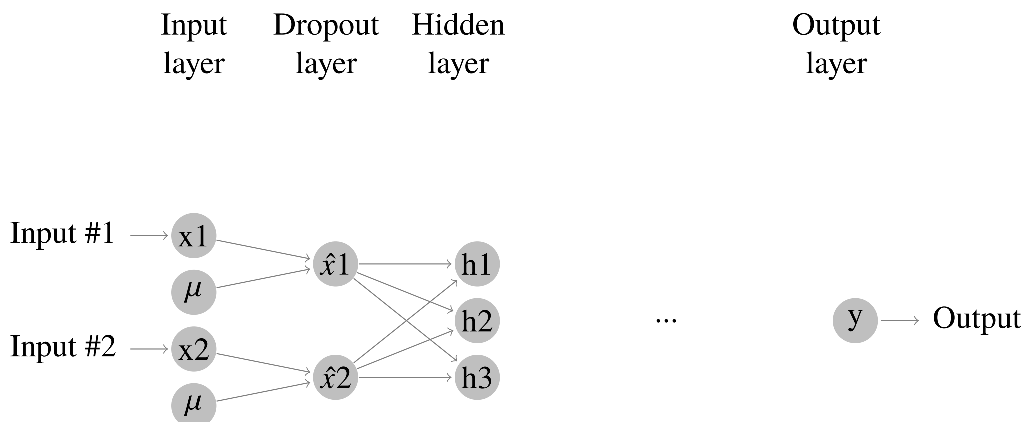


Figure 2.9: An example of a dropout, μ is probability that is a multiplied with the the inputs $x1$ and $x2$.

used for inference (including testing) dropout is not used. Dropout is an effective regularization concept which prevents overfit (Chollet, 2017).

Batch normalization

In machine learning, the concept of normalization is used to make the data samples passed to the model be more similar to each other. In this context, the most common form of normalization is where you center data on 0. That can be achieved by first subtracting the mean of the data, and dividing the difference by the standard deviation of the data. With X being the data, μ being the mean and σ the standard deviation. Then the normalized data, z, is calculated by:

$$z = \frac{X - \mu}{\sigma} \tag{2.12}$$

Applying this normalization on a batch of data, as well as scaling and shifting the data, gets you the full definition of batch normalization (Chollet, 2017; Ioffe and Szegedy,

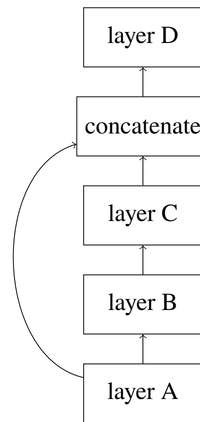


Figure 2.10: A residual connection; the output of layer A & Cj is concatenated.

2015):

$$\mathbf{BN}_{\gamma,\beta} = \gamma z + \beta \tag{2.13}$$

Residual connection

A residual connection consist of concatenating two data representations. This is shown in Figure 2.10 where a previous data representation, that of layer A, is concatenated with a data representation from a layer deeper into the network, layer C. This helps to prevent information loss in the network (Chollet, 2017).

2.4 Architectures

In this section, various state of the art architectures will be explained. The architectures that will be described here are selected on the basis of how they performed on the ImageNet dataset which is shown in Table 2.1.

To increase the accuracy of neural networks, a lot of work has been put into making networks that are deeper and more complicated, but not necessarily making them more efficient with respect to size and speed. The Mobilenet family of neural networks is designed to be more efficient (Howard et al., 2017). Inception (published 2014) is also an architechture which was developed in this style. In fact, it had 12x less parameters than the winning entry of ILSVRC 2012, yet achieving a higher accuracy (Szegedy et al., 2015a).

MobilenetV1

What is distinctive for the mobilenet architectures is the depthwise separable convolution. It is composed of 13 blocks (or 26, counting the parts separately - see Figure 2.11) in succession. They either have stride 1 or stride 2 for their convolution and are surrounded by one fully connected convolution in the front, and a softmax classifier at the end. Each layer (excluding the last) is followed by a batch normalization and a rectified linear unit. As

Architecture	Number of Parameters [Millions]	Operations [Millions]	Accuracy [%]
MobilenetV2_0.25	0.68 (approx.)	27	n/a
MobilenetV1_0.25	0.47	41	49.8
MobilenetV1_0.5	1.34	150	63.3
MobilenetV2_0.5	1.95	97	65.4
MobilenetV1_0.75	2.59	317	68.4
MobilenetV2_0.75	2.61	209	69.8
MobilenetV1_1.0	4.24	569	70.9
MobilenetV2_1.0	3.47	300	71.8
NASnet-Mobile	5.3	564	74.0
MobilenetV2_1.4	6.06	582	75.0
InceptionV4	~ 25 (approx)	5000	80.2

Table 2.1: Number of Parameters - learnable weights, Number of Operations [E.g. Multiply-accumulate (MAC)] and Accuracy of various architectures on Imagenet dataset. For MobilenetV2_0.25 no accuracy on ImageNet was found.

can be seen in Table 2.1 MobilenetV1 comes in a few variants. The decimal at the end of each name is what value a hyperparameter called *width multiplier* is provided. The *width multiplier* is used to scale the number of input/output channels in a certain layer. There is also another hyperparameter called *resolution multiplier* which is used to set input size. The models displayed in 2.1 all use a fixed *resolution multiplier* of 224 (Howard et al., 2017).

In Figure 2.11, a comparison between a conventional convolutional block and the block used in the MobilenetV1 architechture is visualized.

MobilenetV2

In the MobilenetV1 architecture each convolutional operation is followed by a ReLU operation. This is necessary to introduce non-linearities to the network. By definition, the ReLU operation also discards information. In the construction of the MobilenetV2 architecture this is considered. The team behind this architecture shows that if the input can be embedded into lower-dimensional subspaces of the activation space, then the ReLU transformation preserves this information. This is captured by including linear bottlenecks in the MobilenetV2 architecture. It is also empirically shown that non-linear bottlenecks will harm performance, thus the choice of a linear bottleneck. A residual connection is also new to the MobilenetV2 architecture, which is shown in Figure 2.12 (Sandler et al., 2018).

InceptionV4

InceptionV4 is one of the architectures that has achieved the highest accuracy on ImageNet (See Table 2.1). With its new framework (new for this version - Tensorflow) this architecture becomes free from inherent constraints of previous versions, enabling architecture simplifications. Thus it also consists of more inception blocks (Szegedy et al., 2016).

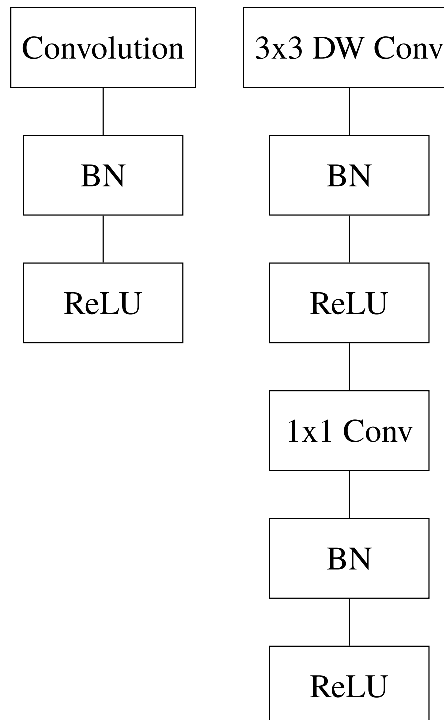


Figure 2.11: Convolutions blocks. Left: Conventional convolutional layer. Right: Depthwise separable convolutional layer followed by convolutional layer, as used in Mobilenet. Abbreviations: DW Conv; Depthwise Convolution, BN; Batch Normalization & ReLU; Rectified Linear Unit.

The Inception-*X* blocks in Figure 2.13 are from earlier versions of Inception. In general, these blocks distinguish the Inception architectures from other architectures.

The simplest of these blocks can be seen in Figure 2.14 which was what the first Inception architecture was based on. This block was revised in Szegedy et al. (2015b), resulting in the three types of blocks that InceptionV4 consist of (Inception-A, Inception-B, & Inception-C in Figure 2.13). The block called "stem" is a special block due to it being close to the input.

NASNet

NASNet is an architecture which was created in an automated process (Zoph et al., 2017). The search method called Neural Architecture Search (NAS) makes use of a control neural net to find the best convolutional architecture for a given dataset. The search for the best convolutional layer (refered to cell in the paper) is done on small dataset (CIFAR-10) first due to cost if applied on a large dataset. Subsequently, the layer is transferred to a larger dataset (ImageNet) where copies of the layer are stacked together (each with their own parameters) to form a convolutional architecture. Two types of convolutional cells were used in the architecture. The normal cell returns a feature map of the same input and the reduction cell returns a feature map where the height and width of the feature map is reduced by a factor of two. The architecture for CIFAR-10 and ImageNet can be seen

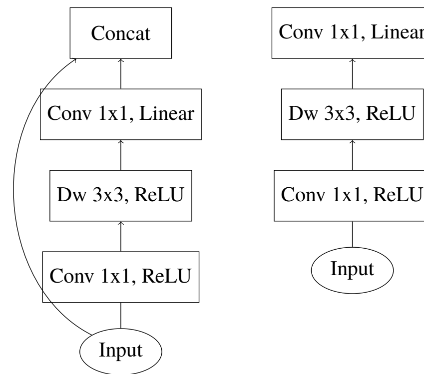


Figure 2.12: The two blocks that are used in MobilenetV2. The operation followed by its activation are shown for each layer in the block. Compare with 2.11. Left: Stride=1 block, Right: Stride=2 block. Abbreviations: DW; Depthwise Convolution, Conv; Convolution, Concat; Concatenation & ReLU; Rectified Linear Unit.

in Figure 2.15. For NASNet-mobile, the normal cells are stacked 12 times between the reduction cells.

2.5 Transfer learning

We use a method called Transfer learning, to transfer knowledge from one solved problem to a different but related one. To do that, we used models which were already pre-trained on the ImageNet dataset. A pretrained model can give advantage if it has been trained originally on a large and general dataset, that is, you do not need to train a network entirely from scratch. Usually, a pretrained model is preferred when there is no big dataset available. Those networks can be seen a a generic model of the visual world, because features learned by the network can be practical for various computer vision problems. We employed two ways of transfer learning: feature extraction and fine-tuning.

- Feature extraction: This technique removes the last fully connected layer and uses the rest of the network to extract features by training a new classifier on top of the "frozen" network (Chollet, 2017).

- Fine-tuning: Instead of freezing the whole network (except the fully connected layer), fine-tuning does unfreeze some of the last layers and training is done on the classifier and the unfrozen part.

2.6 Ensemble learning

Ensemble learning is an approach to create a set of classifiers to solve the same problem. An ensemble consists of a number of independently trained classifiers (e.g. neural networks) called "base learners", whose outputs (e.g. predictions) are combined to classify new data. The ensemble method is mainly used to improve performance and accuracy

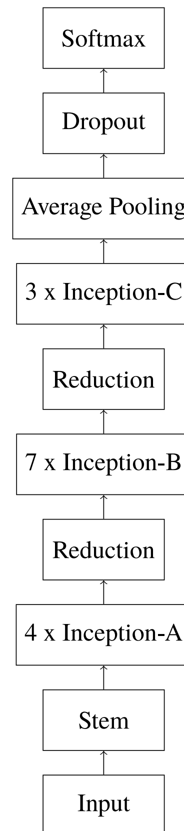


Figure 2.13: Overview of InceptionV4 architechture.

compared to any of the stand alone models. It has been shown that an ensemble usually outperforms each single classifier in the ensemble (Opitz and Maclin, 1999). In this project, we have focused on an ensemble method called *stacking*. Stacking combines a number of individual learners/models whose outputs are used to train another classifier called a meta-learners. We begin by training the models using a training set. Then another distinctive dataset is used as an input to the models to obtain the output which is used as input to train the meta learner. The main idea is that some classifiers could learn features incorrectly, thus misclassifying data. Other classifiers in the ensemble, which are better at classifying (the previously mentioned features) can be used to correct the classification (Opitz and Maclin, 1999; Zhou, 2012).

2.7 Framework

There are quite a few different machine learning frameworks suitable for deep learning models. We considered three different frameworks: *Tensorflow*, *Keras* & *Nnabla*. All three frameworks can be used with different programming languages. Our previous experience with machine learning comes from working with *Python*, hence we decided to adopt it. All frameworks described below support accelerated execution.

- **Tensorflow:** Tensorflow (tf) is an open source software library for machine learning computations, and is mainly designed for deep neural network models. It was

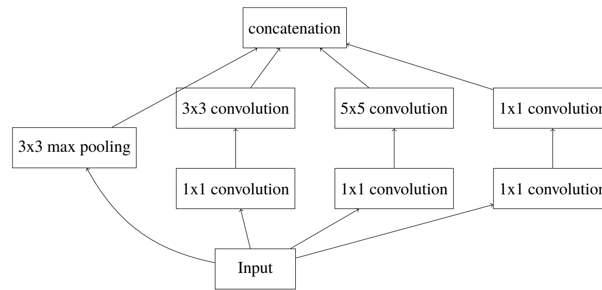


Figure 2.14: The Inception block, containing various transformations which are then concatenated.

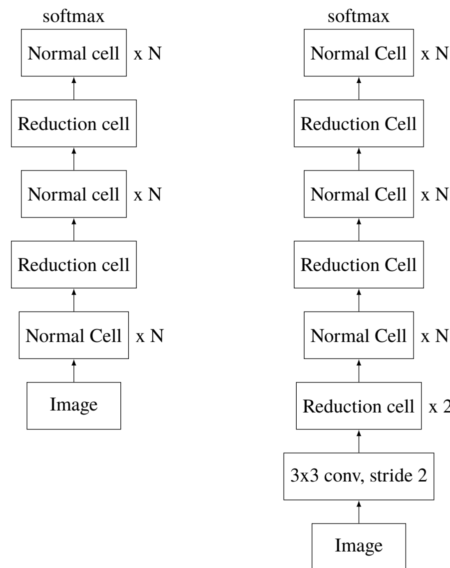


Figure 2.15: Left: Model architecture for CIFAR-10. Right: Model architecture for ImageNet. For NASNet-mobile, N=12

also designed to be a deep learning solution for mobile platforms. Tensorflow is developed by Google Brain at Google, and adhere to a programming paradigm called dataflow programming. That is achieved by first defining a *Graph*. When the *Graph* has been defined, it can be fed with data and executed in a *Session* (Google, 2018).

- **Keras:** Keras is a high level API for deep learning. It requires to be loaded onto a back-end. Eligible back-ends are: *Tensorflow*, *Theano* & *CNTK* (Keras, 2018).

- **Nnabla:** Nnabla is open source software developed by Sony. Python is used for prototyping and experimentation, whereas C++ is used for deploying models to embedded systems (Sony, 2018).

It was important for us that the framework we chose provided extensive documentation. As Tensorflow has a large user base and was already known to us, we decided to adopt it. It also seemed to be the framework with the most compatibility for mobile devices, which also was important to us when selecting an appropriate framework. Tensorflow also includes a high-level API.

We have used *Scikit-learn* to generate the metrics, which are discussed further in Section 3. *Scikit-learn* is also a python framework for machine learning. Tensorflow does not offer the same capabilities in this regard, and would have required us to instead manually implement some of the functionalities that we required.

2.8 Data sources

As the potential of neural networks was realized for computer vision tasks, many datasets were published by different organizations. There is the "Hello World" equivalent of machine learning, MNIST (Modified National Institute of Standards and Technology), which contains handwritten digits with the intended use to classify the digits correctly. Another recognized dataset is the iris flower dataset (FISHER, 1936). Some of the published datasets are related to a computer vision challenge.

Places

The Places dataset was used in the Places2 Challenge, which focused on scene recognition. This dataset contains more than 10 million images spanning more than 400 labels. We have used a variant of this dataset called *Places365-Standard*, which is split into a training, validation and test set (Bolei Zhou, 2017).

ImageNet

Similar to the Places dataset, the ImageNet dataset has also been published for a computer vision challenge, the *ImageNet Challenge*. Rather than scene detection, it focus on object detection and consists of more than 14 million images and more than 20000 labels (We have employed the whole dataset for transfer learning only) (Olga Russakovsky, 2015).

Open Images

The Open Images dataset comes in a few increments and consists of both image label and bounding boxes (object location annotations). It is split into a training, validation and test set. It consists of more than 9 million images with 5000 labels (Krasin et al., 2017; Ope, 2017).

flickr

flickr is an application for photo management and sharing. It has a large user base with an abundance of images and provides an API to interact with the data available on the application. In the API, the functionality to download images with a keyword search is provided.

Other Datasets

The following datasets were all also used in the baseline datasets.

- **CalTech 10'000 Web Faces:** This is a dataset consisting of more than 10000 faces in about 7000 images (Fink, 2007).
- **Stanford 40 Actions:** This dataset was used for human action recognition and its images represents humans performing activities. It consists of more than 9000 images labelled as 40 different activities (B. Yao and Fei-Fei, 2011).
- **University of Insubria's Barcode collection:** This dataset consists of 364 images of barcodes (of insubria, 2016).
- **University of Hong Kong's Blur Detection Dataset:** This dataset was used in a study to find which features of an image can be used to effectively differentiate between blurred and unblurred image regions. It consists of 1000 images (Jianping Shi, 2014).

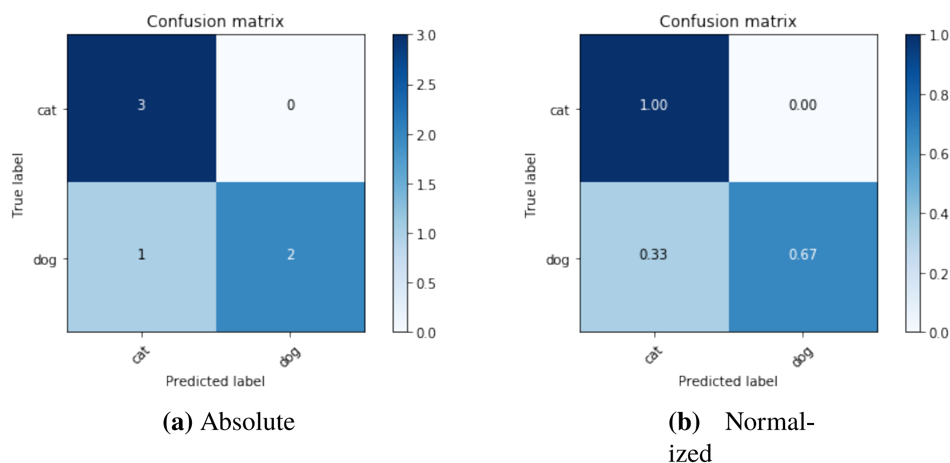
Chapter 3
Metrics

3.1 Confusion Matrix

A *confusion matrix* shows the accuracy of a classifier for each label, and also allows indentifying problematic labels. A confusion matrix is generated after evaluation. An example can be seen in Figure 3.1, where true labels are plotted against predicted labels. Entries on the diagonal correspond to correctly predicted labels. That means that the sum of each row corresponds to the amount of images carrying that label, and the sum of each column is the amount of images that were predicted for that label. In the figure, there are three cat images and three dog images. All of the cat images were correctly predicted as cats (3 in absolute and 1 in normalized confusion matrix). On the other hand, one dog image was predicted as a cat and the rest as dog (1 and 2 in absolute and 0.33 and 0.67 in normalized confusion matrix). A confusion matrix makes it possible to evaluate the performance for each label separately, accounting for uneven distributions in the validation set.

3.2 F1-Score

From the python library *scikit-learn* we use the module *metrics* to calculate a metric called $F1$ (Sklearn, 2017). $F1$ is a harmonic mean of two other metrics, $Recall$ and $Precision$, which are defined as true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). True positives are predictions that are labeled correctly and true negatives are correctly not predicted. False positives are "false alarms", predictions that are incorrectly labeled. False negatives are "missed" predictions, predictions that are incorrectly not labeled. Intuitively put, precision and recall takes two different perspectives, where precision is from the model and recall is from the samples. Thus, precision is a fraction of correct predictions, and recall is a fraction of how much of the ground truths that were



(a) Absolute **(b)** Normalized

Figure 3.1: A small example of a confusion matrix

		Actual	
		Positive	Negative
Predicted	Positive	True Positive	False Positive
	Negative	False Negative	True Negative

Figure 3.2: Precision and recall

captured. Precision and recall can be visualized in Figure 3.2. The definitions of these three metrics as well as *accuracy* are shown in equations below (Goodfellow et al., 2016). Accuracy is the fraction of true labels and the total of labels.

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}} \qquad \text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{Total}} \qquad \text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.3 Evaluation graphs

To visualize our result, we plot a score on the y-axis and epoch on the x-axis. The score can either be an averaged F1-score (where a mean for all classes are used), per-label F1-score or accuracy. Both the scores for the training and validation data are shown. We are mostly interested in how a model performs for the F1-score, as it is a more comprehensive metric. A graph with the F1-score for each class is also used, showing the score for each class seperately. These graphs helps us draw conclusions about the dataset, that is, if more training is necessary and if there is a bias or variance error. The performance of a model can be shown in graph called *learning curve*. It displays how the score is affected by

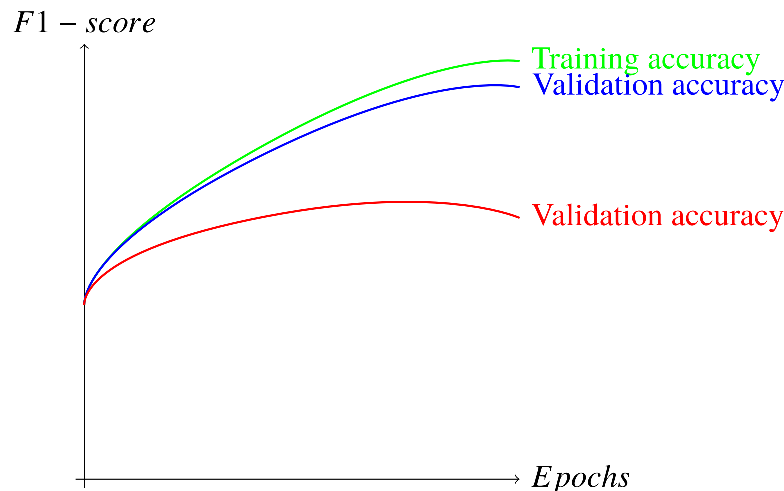


Figure 3.3: Red curve is overfit, the beginning of blue curve is underfit

varying the number of training examples seen by the model. The score is on the y-axis and the number of training examples is on the x-axis. There are two curves in the graph that shows the score for training set and the validation set. If both the curves are reaching a plateau, more training examples would not benefit the model.

3.4 Underfit and Overfit

An important step when training is to evaluate the relation between validation and training F1-score in a graph. The graph gives us understanding whether the model is *overfitting* or *underfitting*. Underfitting means that the model is not capable of understanding the relation of the input data and the corresponding target values. Overfitting is when the model has learned the pattern of the data it has seen but is not able to perform well on data it has never seen before. In Figure 3.3, there are two curves under the training accuracy (red). In the beginning, the green and blue curve is very close to each other which tells us that the model underfitting. A cue is that it is performing bad on both the training set and validation set. This particular figure shows that there is still room for improvement because it has not modeled all pattern of the data which can be seen as the graph is still going up. Training more epochs could solve underfitting. The gap between the green curve and red curve shows that there is an overfit on the model. This means the model is performing good on the training set but poorly on the validation set. Collecting more data could solve overfitting (Chollet, 2017).

3.5 Stratified sampling

Stratified sampling was used when generating the learning curve. It is a method of sampling from a dataset, which is beneficial if number of the samples in categories vary. The train and test sets have the same percentage of samples for each category. For instance, a

category representing 10% of the train set would also represent 10% of the test set.

Chapter 4
Method

4.1 Model

The Tensorflow git repository contains a multitude of resources useful for working with machine learning. Provided are various types of models, categorized as official, research, sample or tutorial, as well as tools to train and evaluate them. Also, access is given to multiple state of the art architectures and comparisons between research models. All the networks that are a part of the comparison are trained on the ImageNet dataset. The networks are compared in terms of accuracy, size and the number of multiply-accumulate (MACs) operations needed to perform inference (see Table 2.1). The architectures as well as the parameters achieved through training are all publicly available. In this project we have utilized both. The architectures available in this repository were used for the various networks that was evaluated. The parameters available were used in the pre-trained models for transfer learning. We have focused on smaller networks, which are assumed to be more suitable for mobile platforms.

4.2 Preprocessing

Before feeding the data to our networks we performed different preprocessing steps: downsizing of the images, converting images to *TFRecords* and data augmentation. Downsizing is a fairly straight forward process, where the resolution was downscaled but the aspect ratio was preserved. This was achieved by the python package called *Python Imaging Library (PIL)*. In that package there is the option to create a thumbnail of a specified image by the provided *thumbnail* method. This method comes with various resampling filters, which if not specified defaults to a bicubic filter (Pillow, 2016). The bicubic filter was the filter that we used in this preprocessing step.

With Tensorflow's data API, it is possible to construct an input pipline to process large

dataset that otherwise wouldn't fit in memory. For this purpose the recommended format is *TFRecords*, which is a binary file. To convert data to this format, the data has to be loaded into a buffer, which is then serialized and where following the output string is written to *TFRecord file* (Tensorflow, 2018).

As is explained in Section 3.4, overfit is a prevalent problem in machine learning. Data augmentation is one way of facing this issue, it's a concept of artificially constructing more data than what is at hand. More data is conctructed by random transformation of each data sample creating similar looking data (but different) to its original (Chollet, 2017). Data augmentation was used on our data by color distortion and cropping.

4.3 Experiment

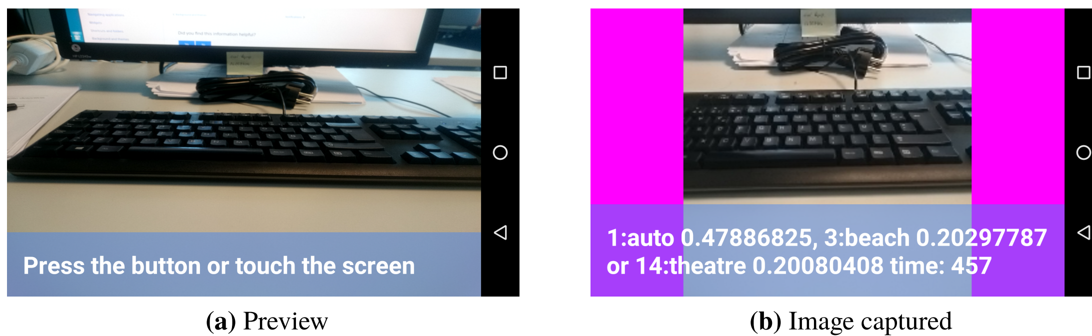
During this thesis, experiments were done in order to answer the research questions. The purpose of the experiments were to find out which architecture to use and whether the model should be a pretrained model (if there exists one) or a model trained from scratch. After training a network, we evaluated the F1 score for each of the categories for all the epochs. We also looked at how well each category performed on the best epoch of the model. We chose the network that had the best average F1-score and did further improvements, such as dataset modifications. After a dataset modification, the network was re-evaluated. For the dataset modifications, we examined whether the categories contained enough images and how meaningful they were. We also investigated if using an ensemble method improves the F1-score. Two ways of transfer learning and optimizers were evaluated. We investigated how well a model trained on the Places dataset could perform on our dataset. Lastly, we investigated how well our model performed in real life using a mobile device.

4.4 Android app

An Android app was developed because we wanted to evaluate the performance of the architectures on the mobile device. The app was primarily used to extract the time it took to run inference on one image. The trained models must be prepared first before deploying them to Android. The preparations are the following:

- Freezing: During training, separate checkpoints are generated which contain the weights of the model. When they are initialized, the latest values are loaded by Variables ops. However, it is not appropriate to have separate files when deploying to mobile device. Instead what we do is called freezing, which load a Graph and retrieve the variables from the latest checkpoint. Then, it will output a Graph where every Variable op is replaced with a Const op which will contain the values of the variables. The script *freeze_graph.py* by Tensorflow was used.

- Optimizing: There are some computations done by the network during training which is not needed when running inference. Parts of the graph can be removed then, to make the network optimized. For instance: Training-only operations like checkpoint saving can be removed and parts of the graph that are never reached/used can also be removed. The script *optimize_for_inference.py* by Tensorflow was used.



(a) Preview

(b) Image captured

Figure 4.1: Android app

- Converting: Lastly, the optimized frozen graph is converted to a Tensorflow Lite Flatbuffer which perform inference. Tensorflow Lite is a lightweight solution for mobile devices from Tensorflow. It allows inference with low latency and small binary size.

The best performing models of all the evaluated architectures are then exported to Android for inference. The Android app captures an image from the camera when the user presses the capture button. The Android app is shown in 4.1. The image is then used as an input to the model for inference. The output of the model are shown in the interface of the app. They are described below:

- Probabilites: The top 4 predictions. The four classes which performed best are shown together with their respective probability.

- Time: The time it took for the model to run inference on one image in milliseconds.

4.5 Data

For this project we had to collect data that would fit to the problem that is to be solved by a neural network. Since the problem was to classify a certain image to a pre-defined scene, we had to create a dataset that had labels which corresponds to those scenes. Through discussions with our supervisor we decided to base those labels on the various scenes that are briefly defined in the Android Camera2 API (developer.google, 2018). In the following, a description of how our dataset was constructed will be provided. We constructed two baseline datasets, which we refer to as Dataset500 and Dataset5000. Dataset500 was mostly used for our own practice, to get acquainted to the framework, whereas Dataset5000 was used for our comparison of networks.

Our Dataset

We have based our dataset on the scenes supported by the Android Camera2 API which are listed in Table 4.1.

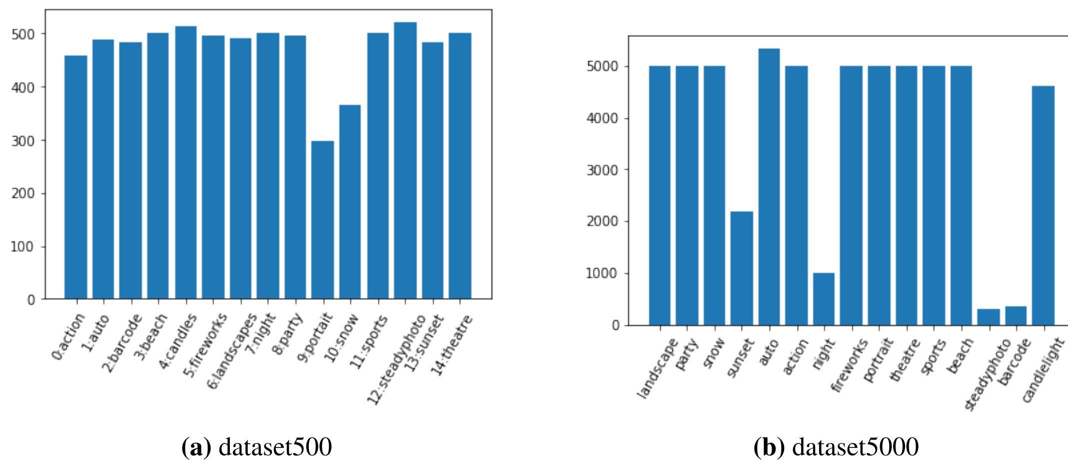
Scene	Definition
Action	Take photos of fast moving objects.
Auto	Scene mode is off.
Barcode	Applications are looking for a barcode.
Beach	Take pictures on the beach.
Candles	Capture the naturally warm color of scenes lit by candles.
Fireworks	For shooting firework displays.
HDR	Capture a scene using high dynamic range imaging techniques.
Landscapes	Take pictures on distant objects.
Night	Take photos at night.
Night Portrait	Take people pictures at night.
Party	Take indoor low-light shot.
Portrait	Take people pictures.
Snow	Take pictures on the snow.
Sports	Take photos of fast moving objects.
Steadyphoto	Avoid blurry pictures (for example, due to hand shake).
Sunset	Take sunset photos.
Theatre	Take photos in a theater.

Table 4.1: The definition of the scenes from the Android Camera2 API (developer.google, 2018)

These definitions works as guideline to what the images included in each label should be representations of. There is some overlapping between certain labels. The problems we faced are the following:

- *Night Portrait*, *Party* and *Portrait* should all contain peoples, but in different light conditions. Due to that, finding images that fits the definition of *Night Portrait* was difficult, and we decided to exclude that label.
- *Action* and *Sports* have the same definition. We made a distinction between them by having *Sports* contain all sport events and *Action* contain everything else that still fits the definition. Detecting fast moving object is more of a video processing problem, but blur is often a cue (Rozumnyi et al., 2016). Hence *Steadyphoto* also overlaps with *Action* and *Sports*. In the baseline datasets blurry images were included in *Steadyphoto*.
- *Auto* contains images that won't fit any of the other used labels, hence it is treated a bit differently (see below).
- *HDR* isn't applicable to the way we have used these scenes, as it is a camera operation mode rather than a scene.

Each label (except *Auto*) was directly mapped from their original dataset, as shown in Table 4.2. For *Auto*, n images per label were extracted from the Places dataset. n was chosen so that a uniform distribution was achieved. Out of the 365 labels which Places consists of, 32 labels were excluded for *Auto*. All the excluded labels can be found in



(a) dataset500

(b) dataset5000

Figure 4.2: Data distribution of the two baseline datasets.

Table 4.3. Some labels were already mapped to other labels in our dataset, while others were too similar to already-mapped labels (e.g. landscape has many similiar labels such as badlands, fields, glacier, etc.). The search for overlapping labels were done manually by us.

Table 4.2: The amount of images for each label and the total amount for the two baseline datasets. From which data source the images were retrieved from are also provided (see section 2.8).

Label	Dataset500	Dataset5000	Original Data source
Action	496	5000	Stanford
Auto	501	5328	Places
Barcode	364	364	Barcode
Beach	500	5000	Places
Candles	484	4615	OpenImages
Fireworks	484	5000	OpenImages
Landscapes	520	5000	Places
Night	514	1008	OpenImages
Party	488	5000	Places
Portrait	496	5000	CalTech
Snow	459	5000	Places
Sports	500	5000	Places
Steadyphoto	296	296	Blur
Sunset	491	2184	OpenImages
Theatre	500	5000	Places
Total	7093	58795	-

Scene in Places	Label in our dataset	Scene in Places	Label in our dataset
Beach	Beachb	Igloo	Snow
Badlands	Landscape	Ski slope	Snow
Field - wild	Landscape	Ski resort	Snow
Field - cultivated	Landscape	Snowfield	Snow
Forest path	Landscape	Classroom	Theatre
Golf course	Landscape	Movie theater	Theatre
Hayfield	Landscape	Stage - indoor	Theatre
Lagoon	Landscape	Stage - outdoor	Theatre
Mountain	Landscape	Bar	Party
Mountain path	Landscape	Discotheque	Party
Ocean	Landscape	Pub	Party
Pasture	Landscape	Boxing ring	Sports
Tundra	Landscape	Baseball field	Sports
Valley	Landscape	Basketball court	Sports
Glacier	Snow	Soccer	Sports
Soccer field	Sports	Volleyball court	Sports

Table 4.3: The labels that were excluded from Places when constructing *Auto*.

Chapter 5
Results

All models were evaluated for 50 epochs, except when the average F1-score plateaued or the gap between training and validation F1-score increased (overfitting). A batch size of 32 was used for all architectures except NasNet Large, which used a batch size of 8. This section will present our result.

5.1 Comparison of networks

The comparison between the different networks can be seen in Table 5.1. The architecture InceptionV4 achieved highest F1-score (78%) when trained from scratch. When feature extraction was done, Mobilenet-v1 (80.6%) achieved the highest F1-score among the architectures. This model achieved the highest F1-score among all the evaluated models. Across all models feature extraction performed better than when trained from scratch. Additional results are provided for the highest performing models for both feature extraction and training from scratch in the following.

In the column "Time" in Table 5.1, each time was calculated from an average of 7 measurements. As can be seen, the Mobilenet v1 & v2 have inference runtimes on par with corresponding depth-wise multipliers. See Table 2.1 for number of operations for each model. The inference was executed on a Sony Xperia Z3.

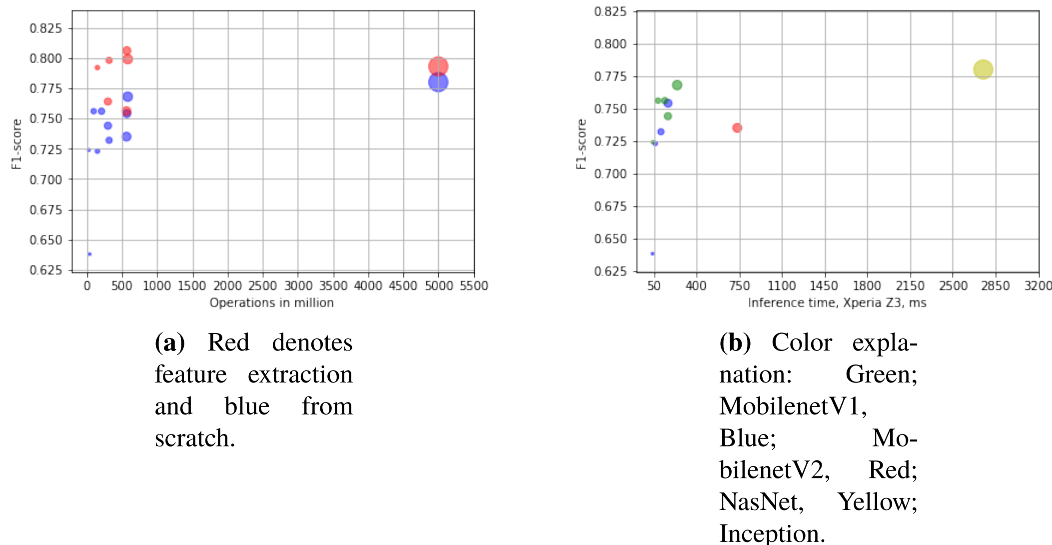
5.1.1 InceptionV4

In Figure 5.2 it is clear that there is a steady increasing improvement, but at a point in between epoch 25 and 35 the gap between training score and validation score starts to increase. This suggest the presence of an overfit on the training data.

Across all models that were evaluated, *steadyphoto*, *night*, *auto* and *action* had the lowest F1-score. This can be seen in Figure 5.3 and is true for the other models as well.

Architechture	F1-score (from scratch)	F1-score (feature extraction)	Time for inference (ms)
Mobilenet-v1	0.754	**0.806**	168
Mobilenet-v1-0.75	0.732	0.798	110
Mobilenet-v1-0.5	0.723	0.792	64
Mobilenet-v1-0.25	0.638	-	**41**
Mobilenet-v2-1.4	0.768	0.799	243
Mobilenet-v2-1	0.744	0.764	167
Mobilenet-v2-0.75	0.756	-	141
Mobilenet-v2-0.5	0.756	-	86
Mobilenet-v2-0.25	0.724	-	43
NasNet-Mobile	0.735	0.756	735
NasNet-Large	-	-	-
InceptionV4	**0.780**	0.793	2750

Table 5.1: The highest F1-score and corresponding epoch of various architectures. Training was done from scratch or using feature extraction from a pretrained model



(a) Red denotes feature extraction and blue from scratch.

(b) Color explanation: Green; MobilenetV1, Blue; MobilenetV2, Red; NasNet, Yellow; Inception.

Figure 5.1: The size of each dot corresponds to the number of parameters for each architecture.

Consequently, the defects of the previously mentioned labels are present in Figure 5.4. Where, 31% of the *steadyphoto* images are being labelled as *action*, and 12% as *auto*, resulting in only 50% being labelled correctly. *Night* is confused a lot with *fireworks*, where most of the images are labelled as *fireworks*. *Auto* has many of its true images spread out across most other labels, in particular: *action*, *beach* & *candlelight*. These results are consistent with the results of the other models that were evaluated on this dataset.

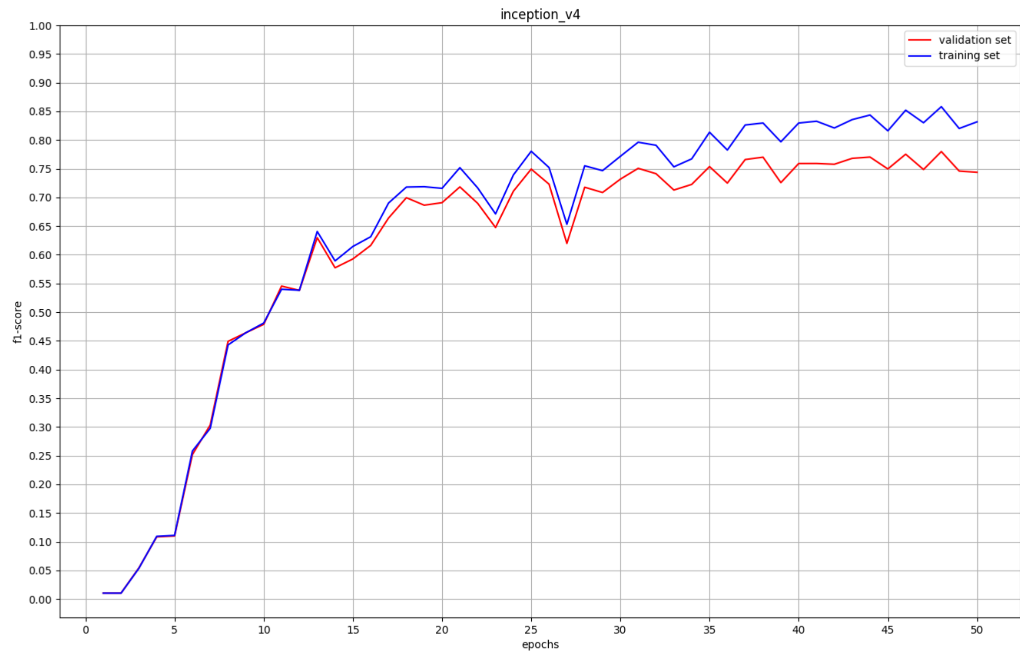


Figure 5.2: F1-score for InceptionV4 trained from scratch.

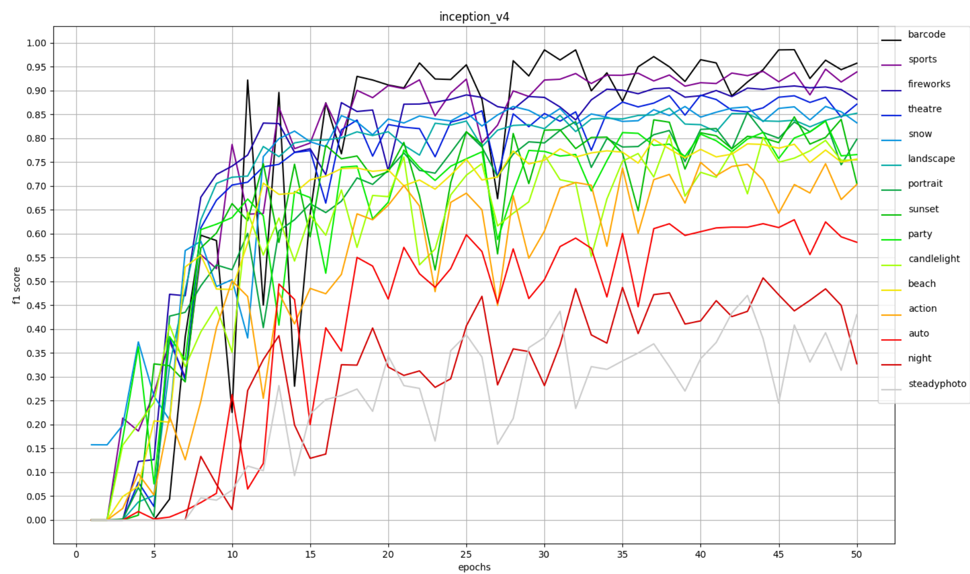


Figure 5.3: F1-scores for each class of InceptionV4 trained from scratch.

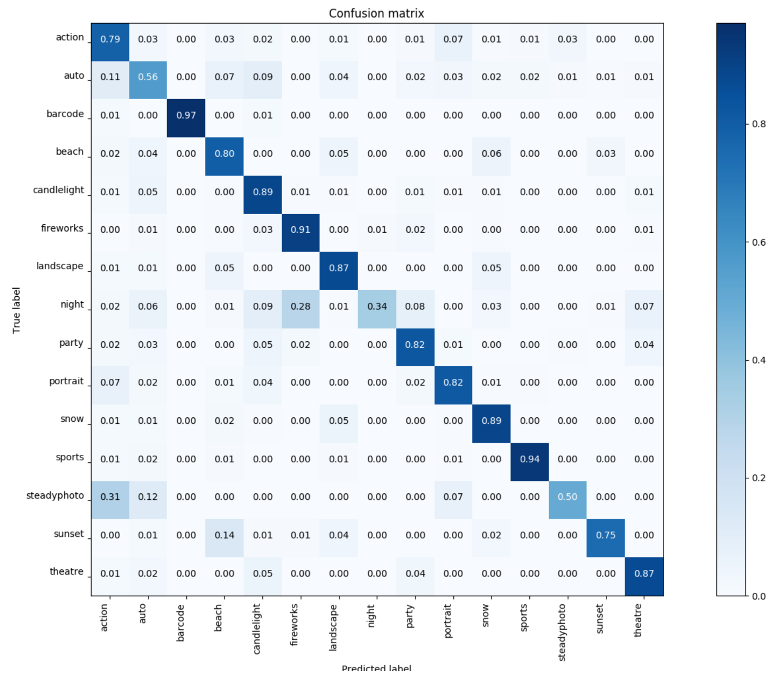


Figure 5.4: Normalized confusion matrix for InceptionV4 trained from scratch.

5.1.2 MobilenetV1

Both of the curves in Figure 5.5 are fairly flat and congruous, hence no big improvements are made on the pre-trained model by feature extraction. All evaluated pre-trained models have the same behaviour for their F1-score.

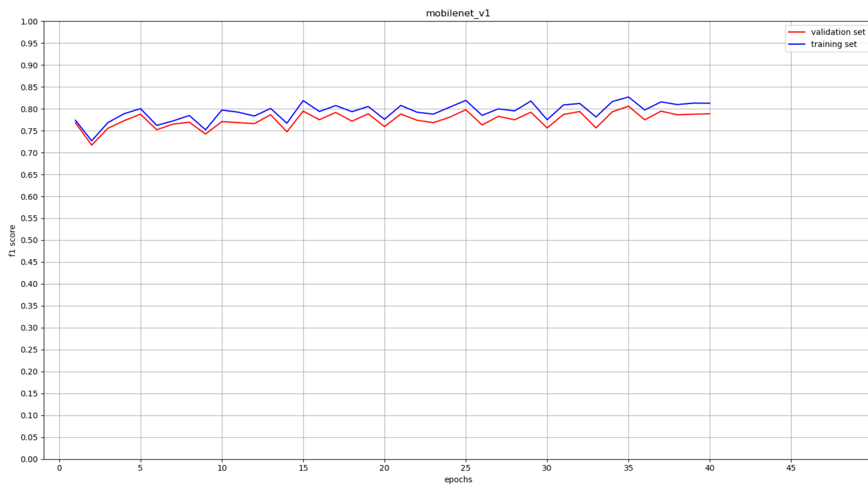


Figure 5.5: F1-score for Mobilenet-v1 pre-trained on ImageNet.

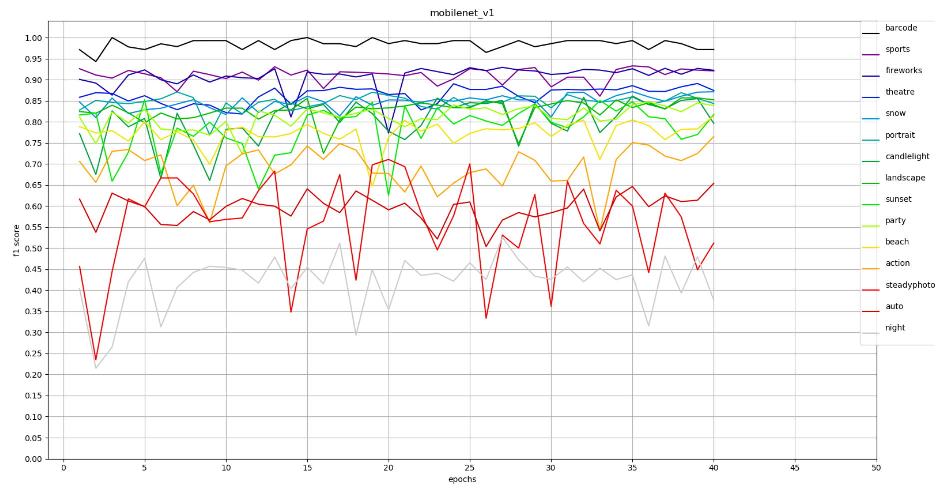


Figure 5.6: F1-scores for each class of Mobilenet-v1 pre-trained on ImageNet.

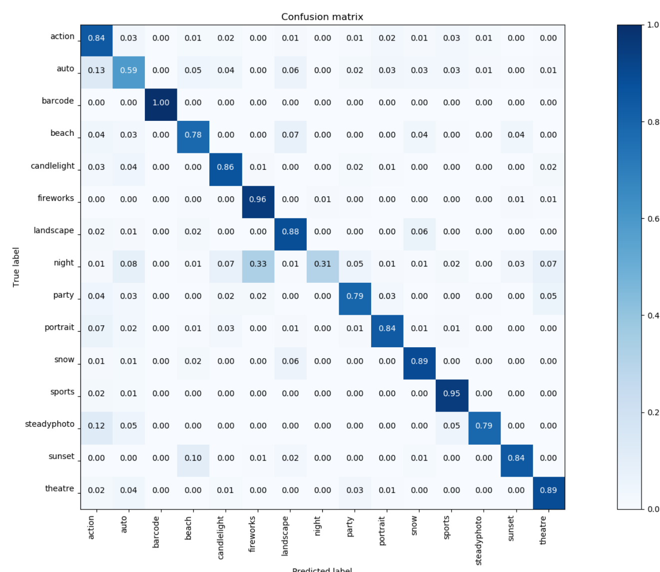


Figure 5.7: Normalized confusion matrix for Mobilenet-v1 pre-trained on ImageNet. See appendix for corresponding confusion matrix with absolute numbers.

Many of the same issues as for InceptionV4 are present in this model as well (Figure 5.6 & 5.7), except for *Steadyphoto*, which shows a big improvement. Also notable is that all images of *Barcode* are labelled correctly, both in terms of precision and of recall.

5.2 Dataset modifications

After comparing the architectures, modifications on the dataset were made in order to improve the results. The most important dataset modification is listed below and the best performed model from the previous result (Mobilenet-v1) was used to train on it. The category *Steadyphoto* was removed and the number of images in the category *Night* was increased. With this modification an F1-score of 84.1% was achieved.

5.3 ADAM and fine tuning

The optimizer was changed to investigate if the F1-score would improve. The dataset which was modified as described in the previous subsection was chosen to train on. This change resulted in an F1-score 84.7%, a small increase compared to the previous (84.1%). Fine tuning was done to see if the result could further improve. The layer before the last layer was also trained. By doing fine tuning, the F1-score increased to 86.9%.

Table 5.2: Performance of ADAM on datasetv1 using transfer learning

Improvement	F1-score
ADAM OPT + feature extraction	0.847
ADAM OPT + fine tune	**0.869**

5.4 Places dataset

The results from pre-training a model on the Places dataset, and feature extraction on dataset5000v1 are provided here. We created the pre-trained model ourselves, based on a modified variant of Places. Places was modified by removing the labels that are used in dataset5000v1(see Table 4.2), except for auto. For auto all images used in dataset5000 were removed. This was done in order to prevent the network from being trained on an images it already has been trained on.

With this model an F1-score of 83.2% was achieved, it can be compared to the result of Section 5.2. Figure 5.8 and 5.9 shows that F1-score is constant throughout the whole training session, with few modifications on the pre-trained model. As can be seen in 5.10, there is a confusion between the labels *sunset* and *beach*. What is also notable is that action "swallows" many of the images this model is provided with.

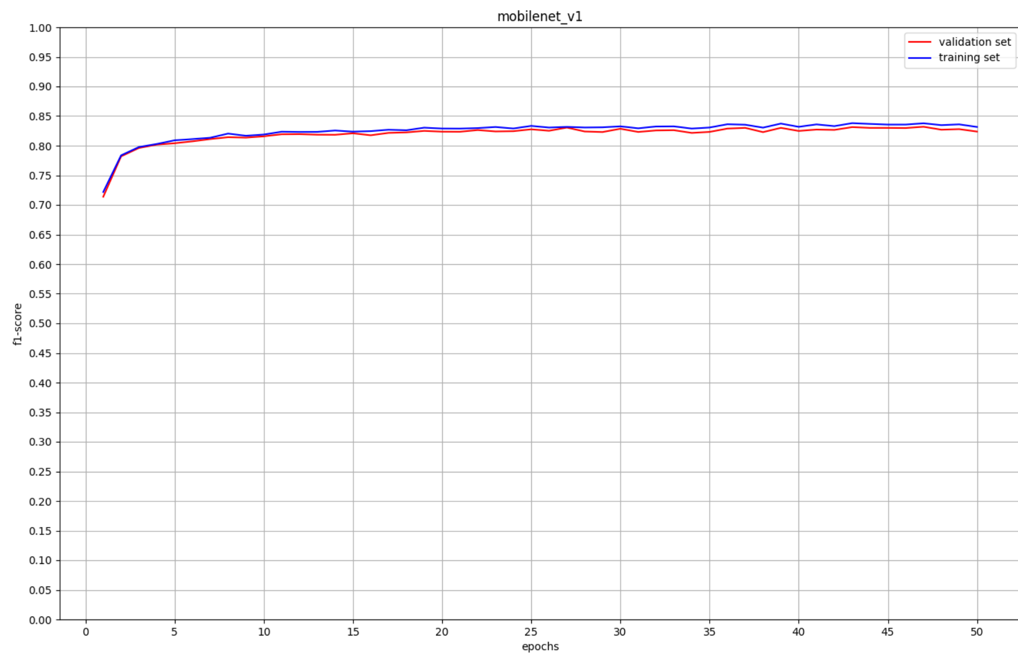


Figure 5.8: F1-score for Mobilenet-v1 pre-trained on places.

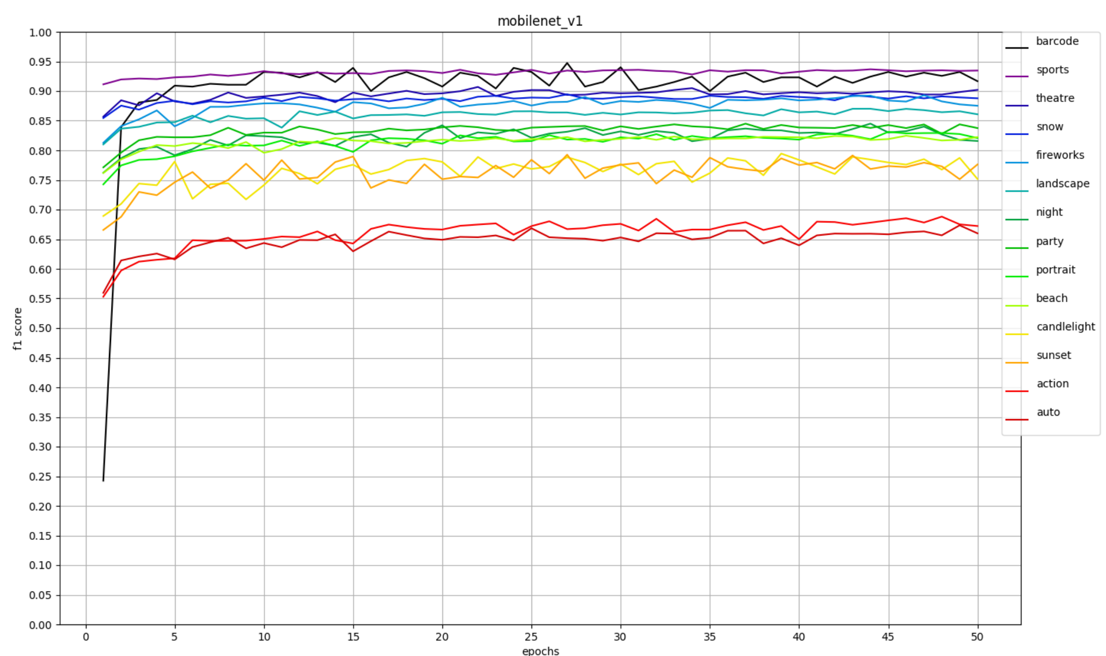


Figure 5.9: F1-scores for the separate classes in Mobilenet-v1 pre-trained on places.

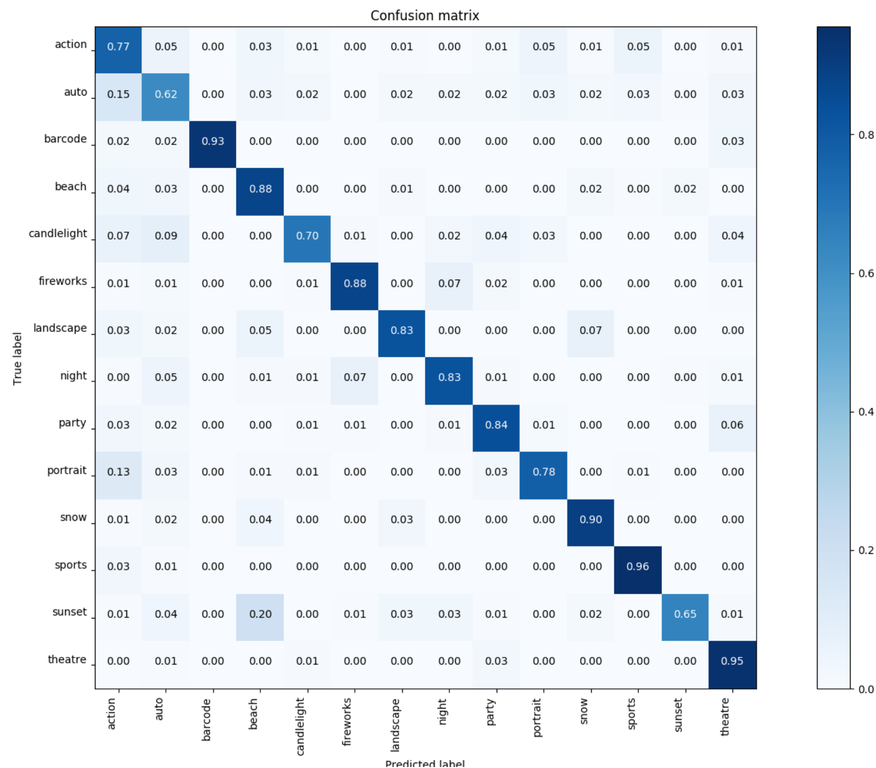


Figure 5.10: Normalized confusion matrix for Mobilenet-v1 pre-trained on places. See appendix for corresponding confusion matrix with absolute numbers.

5.5 Learning curve

A learning curve was created by training and evaluating the model Mobilenet-v1 for each set of all training examples. Each training set contained 4960 images, where the set has a proportional split percentage compared to the validation set by using stratified sampling. The split percentage can be seen in Table 5.3. The total number of training examples seen by the model is 49600 examples. The validation set contains 9920 examples. These examples are extracted from Dataset5000v1 which is not composed of a perfect uniform distribution. Corollary, *Candles*, *Sunset* and *Barcode* has lower percentages in Table 5.3. That had no noticeable effect on the result.

Table 5.3: Stratified sampling.

Model	Percentage	Train size	Validation size
Candles	0.071	3550	710
Sunset	0.035	1750	350
Barcode	0.006	300	60
All other classes	0.08	4000	800

Figure 5.11: Learning curve. A stratified split was used.

5.6 Ensemble learning

The result for the ensemble learning is shown in Table 5.4. There are two ensembles, each containing two different models. Mobilenet-v1 was trained on a size of 36990 images, using similar modifications as was done to Dataset5000v1 (see Section 5.2). Fine-tuning was also done for that dataset. Mobilenet-v1 was also trained on a dataset with the following modifications to the baseline dataset: more images for *Night*, *Sunset* and *Auto* (images were added from the ImageNet dataset to increase diversity). We refer to this dataset as Dataset5000v8. The ensemble containing of the feature extracted (fe) v1 and v8 achieved F1-score of 85%. The ensemble containing of the fine tuned (ft) v1 and v8 achieved a F1-score of 86.6%. The meta-learner was a fully-connected layer where the input was the outputs from both the models (14 predictions each) making it the size of 28.

Table 5.4: Ensemble learning. The architecture Mobilenet-v1 was used. A validation size of 10000 was used for each F1-score.

Model	Train size	F1-score
v1 (feature extraction)	36990	0.845
v1 (fine tuning)	36990	0.865
v8	44719	0.841
ensemble (v1 (fe)+v8)	15493	**0.850**
ensemble (v1 (ft)+v8)	15493	**0.866**

Chapter 6
Discussion

6.1 Comparison of networks

In the comparison of networks trained on ImageNet (see Table 2.1), NasNet-Large had the highest accuracy. The performance then followed in descending order with regard to number of operations. When looking at how the same networks performed on our dataset when trained from scratch, that principle is still fairly accurate. With two exceptions, Mobilenetv2-1.0 and NasNet-Mobile were both expected to perform much better. Looking at how the pre-trained models performed, these two networks are the two with the lowest F1-score. The reason for this is hard to figure out since you generally don't know what networks learn. It could be possible to increase the score of these by individually tweaking them. To have a fair assessment, it would then be required to have all networks individually tweaked. Due to limited time we mainly considered the best performing networks for further tweaking and investigation.

What is also interesting to note is that a small cut in F1-score, will correspond to a huge cut in inference time. As an example, consider Mobilenet-V1-1 and Mobilenet-V1-0.5 in Table 5.1. There is an difference in F1-score of 1%, but inference time takes more than twice as long for the former (168 ms) as the latter (64 ms).

Table 5.1 is sparse because for some architectures, no pretrained models were found. NASNet-Large was omitted, due to slow training.

6.2 Dataset modifications

More images were added to the Night category due to low F1-score according to the class graph for all architectures. The category Steadyphoto was removed because it also achieved a very low F1-score and the complexity and ambiguity led to the decision to try and leave it out. The Android Camera API describe the scene as "Avoid blurry pictures

(for example, due to hand shake)". It is hard to decide based from an image whether it could be classified as a Steadyphoto. There is a difficulty for the network to classify images as Steadyphoto when there could be clashes between this class with sports and action images which could all be a bit blurry but be more suited to sports and action due to the fast movement.

Combining action and sports images was done because the action and sports scenes were both described the same way by the Android Camera parameters as "Take photos of fast moving objects".

6.3 ADAM and fine tuning

By changing the optimizer from RMSProp to ADAM, the model increased its F1-score to 84.7%. This increase could be that ADAM is using momentum which gives a faster convergence. Fine tuning also increased the performance of the model. Due to the features of the last layers are more specialized (Chollet, 2017), it will fit our problem more by training the last layers.

6.4 Places dataset

In (Zhou et al., 2017), they show that there is classification difference for object-centric and scene-centric images. Since most of our images are scene-centric, we expected a higher performance score for the pretrained model on places. Unfortunately it didn't, it achieved a result that was 0.9% points lower than the pretrained model on ImageNet.

6.5 Learning curve

The learning curve revealed that using more training examples would not be necessary. Both the training and validation curves started to plateau at around 44640 examples seen by the model.

6.6 Ensemble learning

Using stacking as an ensemble method gave a slight increase in F1-score when the ensemble contained two models. One model was good at classifying the category auto (v8) and the other one was an all round model (v1) but not good at auto images. However, the new model is not desirable to run on a mobile device considering that it could take as long as 335ms (Mobilenet-v1) to run both (assuming it is run after each other) the base learners and the meta-learner (time neglected). Also, the performance gain would only be 0.1% or 0.5% points.

In the context of convolutional networks and stacking, the choice of an ensemble size may be a trade off between accuracy and inference time. In an ensemble of many networks, it might not be desirable that each network has too long inference time. From the result in Table 5.4, it can be seen that a small increase in accuracy corresponds to a large increase in

operations (see Table 2.1). For mobile devices, it might be preferable to choose a network with lower performance if you can gain processing time, to preserve power consumption. Considering that each model is running inference in sequence, too many models could give a large combined inference time.

Chapter 7
Conclusions & Future work

7.1 Conclusions

In this thesis, we have investigated the possibility of using machine learning to do scene detection. Various convolutional neural networks have been evaluated on our dataset. The networks have been trained from scratch and on pretrained models (feature extraction). We chose to build our dataset by using the scenes defined by Android Camera API as our labels. Metrics were used to measure the performances of the models. An ensemble method and finetuning were explored because it could potentially increase the performance of a model.

We found that the best architecture was MobiletNet-v1-1 (feature extraction). Using a pretrained model proved to be an advantage, though training can probably be cut down a bit cause the F1-score was more or less constant throughout the training session. We also created a pretrained model on another dataset, places, on MobiletNet-v1-1 architecture, but no signifcant improvement was achieved.

Some dataset modifications improved the performance (increasing Night images) but others did not show any significant improvements. Different changes to Auto was made but none gave a better result in the end. We tried to increase the number of images in Auto four times by retrieving images both from ImageNet (object centric images) and Places (scene centric images). While leaving Auto out, the performance of the model increased. This label was the one that caused us the most problem. If possible labels should be constructed in such a way that labels of this kind (acting as other) can be avoided. We also found out that the category Action did not perform well in reality. When a scene contained people, it often said the scene was Action. This is most probably due to the images in the training set for action always contain actions performed by people. We looked into the images in Action closely, and realized that it was not always clear that an action was performed according to the Android definition (fast moving object). There were some badly defined scenes which led to Steadyphoto of them being removed from the dataset.

Finetuning and changing the optimizer from RMSProp to ADAM further improved the result. The ensemble method stacking showed better performance than a separate model. Thus, to achieve the highest performance on our dataset, networks in an ensemble should be employed. With this solution we think it is a fair assumption to make that the inference time will increase proportionately to the networks employed. This is something that has to be considered due to the time constraint inherent for this type of problem.

On that note, you can even make a case for MobiletNet-v1-050 being the most suitable architecture. It has an inference time that is 2x less than MobiletNet-v1-1 while the performance difference is only 1%.

7.2 Future work

We will present some suggestions of how to continue with this work

- Instead of predicting a scene which then maps to a camera parameter setting, a model which bypass scene prediction and instead directly predict a parameter setting would be of interest. This kind of model could provide a spectrum of parameter settings, rather than 15 discreet parameter settings as would be the case for our model.

- This task could be suitable for an application of an Generative Adverserial Network (GAN). The generator network would then take a camera sensor reading as its input, apply camera parameter settings to it and pass it to the discriminator. One problem with this solution is that the discriminator requires a ground truth input which represents "good" camera parameter settings, but what are "good" camera parameter settings?

- Using other ensemble methods. In this thesis, stacking was explored but there are other ensemble methods that could perform even better. Each ensemble method has its own weakness and strength. The size of the ensemble must also be chosen carefully because they have an effect on the accuracy.

- Avoid overlap between scenes by having more clearly defined scenes and construct them in such a way that an "other" scene is avoided. This could be achieved by a multi-scene model, where each scene can be labelled with multiple labels. E.g, an image that contain both a beach and fireworks, would be labelled accordingly as "beach, fireworks".

Bibliography

(2017). Open Images Dataset V3. https://github.com/openimages/dataset/blob/master/READMEV3.md. [Online; accessed 2018-07-19].

B. Yao, X. Jiang, A. K. A. L. L. G. and Fei-Fei, L. (2011). *Human Action Recognition by Learning Bases of Action Attributes and Parts*. Internation Conference on Computer Vision. Barcelona, Spain.

Ballard, D. H. and Sabbah, D. (1983). Viewer independent shape recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(6):653–660.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer.

Bolei Zhou, Agata Lapedriza, A. K. A. T. A. O. (2017). Places2: A Large-Scale Database for Scene. http://places2.csail.mit.edu/challenge.html. [Online; accessed 2018-07-19].

Chollet, F. (2017). *Deep learning with Python*. Manning Publications Co.

developer.google (2018). Android API: Camera.Parameters. https://developer.android.com/reference/android/hardware/Camera.Parameters. [Online; accessed 2018-07-19].

Fink, M. (2007). Caltech 10, 000 Web Faces. http://www.vision.caltech.edu/Image_Datasets/Caltech_10K_WebFaces/. [Online; accessed 2018-07-19].

FISHER, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. [Online; accessed 2018-05-03].

Google (2018). TensorFlow. https://www.tensorflow.org/extend/tool_developers/. [Online; accessed 2018-07-19].

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

Jianping Shi, Li Xu, J. J. (2014). University of Hong Kong: Blur Detection Dataset. `http://www.cse.cuhk.edu.hk/leojia/projects/dblurdetect/dataset.html`. [Online; accessed 2018-07-19].

Karpathy, A. (2018). cs231n cnn. `http://cs231n.github.io/convolutional-networks/`. [Online; accessed 2018-07-19].

Karpathy, A. and Fei-Fei, L. (2017). Deep visual-semantic alignments for generating image descriptions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):664–676.

Keras (2018). keras. `https://keras.io/`. [Online; accessed 2018-07-19].

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krasin, I., Duerig, T., Alldrin, N., Ferrari, V., Abu-El-Haija, S., Kuznetsova, A., Rom, H., Uijlings, J., Popov, S., Veit, A., Belongie, S., Gomes, V., Gupta, A., Sun, C., Chechik, G., Cai, D., Feng, Z., Narayanan, D., and Murphy, K. (2017). Openimages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://github.com/openimages*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

of insubria, U. (2016). Medium 1D barcodes collection. `http://artelab.dista.uninsubria.it/downloads.html`. [Online; accessed 2018-07-19].

Olga Russakovsky, J. D. (2015). ImageNet Large Scale Visual Recognition Challenge. `http://www.image-net.org/challenges/LSVRC/`. [Online; accessed 2018-07-19].

Opitz, D. and Maclin, R. (1999). Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11:169–198.

Pillow (2016). Pillow: Image Module. `https://pillow.readthedocs.io/en/3.1.x/reference/Image.html`. [Online; accessed 2018-07-19].

Rozumnyi, D., Kotera, J., Sroubek, F., Novotný, L., and Matas, J. (2016). The world of fast moving objects. *CoRR*, abs/1611.07889.

Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L. (2018). Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381.

Sklearn (2017). sklearn metrics f1 score. `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html`. [Online; accessed 2018-07-19].

Sony (2018). nnabla. `http://nnabla.readthedocs.io/en/latest/`. [Online; accessed 2018-07-19].

Szegedy, C., Ioffe, S., and Vanhoucke, V. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., et al. (2015a). Going deeper with convolutions. Cvpr.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015b). Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567.

Tensorflow (2018). Tensorflow: File formats. `https://www.tensorflow.org/api_guides/python/reading_data#file_formats`. [Online; accessed 2018-07-19].

Tieleman, T. (2018). RMSPROP. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. [Online; accessed 2018-07-19].

Zhou, B., Lapedriza, A., Khosla, A., Oliva, A., and Torralba, A. (2017). Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., and Oliva, A. (2014). Learning deep features for scene recognition using places database. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 487–495. Curran Associates, Inc.

Zhou, Z.-H. (2012). *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6).

INSTITUTIONEN FÖR DATAVETENSKAP & MATEMATIKCENTRUM | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2018-08-23

EXAMENSARBETE Generation of Artificial Training Data for Deep Learning
STUDENTER Pontus Andersson, David Wessman
HANDLEDARE Michael Doggett (LTH), Kalle Åström (LTH)
EXAMINATOR Niels Christian Overgaard (LTH)

Konstgjorda träningsexempel för artificiell intelligens

POPULÄRVETENSKAPLIG SAMMANFATTNING **Pontus Andersson, David Wessman**

Kan bilder av konstgjorda människor ersätta bilder av riktiga människor som träningsexempel för artificiell intelligens? Vi undersöker denna fråga och skapar ett ramverk för att generera stora mängder konstgjorda träningsexempel.

Säg att vi vill träna upp en artificiell intelligens (AI) för att hitta katter i bilder. Det viktigaste som behövs är träningsexempel. Först behövs en bild på en katt och sen måste en manuellt markera var i bilden katten är. Helst behövs det hundratusentals bilder, några med och några utan katter. Hur löser vi detta?

Vårt förslag är att skapa konstgjorda träningsexempel! Istället för katter har vi tittat på träningsexempel för att kunna avgöra om det är samma människa som syns i två olika bilder. Bilderna genereras med datorgrafik, på samma sätt som i dator- och TV-spel.

När vi testar vår AI, efter att den tränats endast med genererade bilder, är resultatet inte lika bra som när vi tränar på riktiga bilder. Detta kan förklaras av skillnaderna som syns om man jämför bilder vi genererar (**A**) med bilder från riktiga övervakningskameror (**C**). Kan skillnaderna göras mindre? Vi försöker åstadkomma detta genom att sända alla våra egna bilder genom en processor vars uppdrag är att *förfina* våra bilder, dvs. att få dem att se ut som att de också var tagna med riktiga övervakningskameror och innehöll verkliga människor (**B**).

I våra resultat ser vi att konstgjorda bilder inte kan ersätta riktiga bilder som träningsexempel helt och hållet. Däremot upptäcker vi att förfinade bilder utgör bättre träningsexempel än de som inte förfinats.

Om konstgjorda bilder kunde ersätta riktiga, skulle det bli mycket enklare att skapa färdiga och uppmärkta träningsexempel, även för tillämpningar där det idag inte finns bilder eller träningsexempel att använda. Till skillnad från bilder av riktiga personer, är genererade bilder dessutom inte integritetskränkande.

För att kunna generera stora mängder träningsexempel lånar vi våra kollegors datorer på natten. Vi har byggt ett system där alla datorer genererar bilder som automatisk skickas till vår dator. Detta gör att vi kan generera hundratusen bilder varje natt.

Bild (C) från bildsamlingen **CUHK03**.